Embedded Systems with Linux

Programming the GPMC driver

Manuel Domínguez-Pumar

Embedded Systems with Linux series

Volume 3: Programming the GPMC driver

Manuel Domínguez Pumar

Electronic Engineering Department

Technical University of Catalonia – BarcelonaTech

First Edition – July 2018

ISBN 978-84-09-03815-2

AMG Editions, Barcelona, Spain

Artwork: JP Productions

Index

1 Drivers

1.1 Introduction

Drivers make up the layer of programs that have a direct contact with system hardware. We can see intuitively that, for example, in order to write a file in a hard disk it is necessary a series of operations of writing and reading in control and data registers at the disk controller. These operations will depend on the specific hard disk we have. However each hard disk will need its specific operations, completely different and in principle incompatible with the ones needed by any other disk.

To use a unified approach for communicating with the hardware, operating systems provide the concept of 'module' or driver. All OS present an adequate interface common to all peripherals so that it is possible to communicate, at least to some extent, with different hardware using the same interface.

The system devices are at */dev* directory of the Linux or Unix tree. They have a format similar to files and therefore we can do with them usual operations such as *open*, *close*, *read*, *write*, etc.

As an example, in the Beaglebone we can look into the *urandom* device, which provides random numbers:

```
beaglebone# od -x /dev/urandom
```

The *od* command with option *-x* reads the device in hexadecimal.

Drivers are also named ‘modules’. You can load a driver using the command:

```
insmod 'driver name'
```

And we can see the drivers installed at the system with the command:

```
lsmod
```

Execute this command both in the beaglebone and in the host PC.

The command to remove a driver is:

```
rmmod 'driver name'
```

Up to 15 different operations, each one related to a system call, can be used to work with files. Drivers must provide 'an answer' to at least some of them (for example, *open*, *release*, *read* and *write*). Furthermore, each time the module is loaded it is necessary to configure a certain piece of hardware so that later, when an application accesses the module, the peripheral will be correctly configured.

With all this in mind, in the module/driver we must provide some code for the following items:

• **Module initialization**: We must point out to the OS the code providing the initial configuration of the peripheral.

• **Module exit**: Code providing the configuration we want to have in the peripheral once the module has been removed.

• **Normal file operations**: open, release (=close), read, write.

• **Other configurations**, such as *ioctl*. This system call is a way the application has to provide a value to a certain variable. We will see a simple implementation of this later.

In order to pass this information to the operating system, we will generate a structured variable with the following fields:

```
static struct file_operations sd_fops =
{
   write:              sd_write,
   read:               sd_read,
   open:               sd_open,
   release:            sd_release,
   unlocked_ioctl:     sd_ioctl,
};

static struct miscdevice sd_devs =
{
   minor:      MISC_DYNAMIC_MINOR,
   name:       DRIVER_NAME,
   fops:       &sd_fops
};
```

We have defined the variable *sd_fops* which has at least 5 fields: *write*, *read*, *open*, *release* and *unlocked_ioctl*. The values given to each one of these fields (*sd_write*, *sd_read*, etc.) are the **names** of the routines that will be included inside our module and that must be executed once the corresponding system call is made by an application to our driver.

Assuming our module is related to the device */dev/driver_port*, when we execute within a user application:

```
f=open("/dev/driver_port", O_RDWR);
```

we will be executing the routine indicated by the field *open* in the *sd_fops* variable. In our case, this routine is called *sd_open*.

Summarizing, we are mapping the routines in our module to the system calls that can be done on our device.

This operation is completed when executing:

```
misc_register( &sd_devs )
```

This is an specific function that allows to register our device inclosed in the category of 'miscellaneous devices'. This is a simple way of registering our device. The name of our device, namely *driver_port* is contained in the field *DRIVER_NAME* of the variable *sd_devs* that is passed to *misc_register()*.

1.2 A first driver example

The module below (*module.c)* is a very simple example. The only significant code is contained in the '*sd_read*' function, where the ID of the microprocessor in the board is read.

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/miscdevice.h>
#include <linux/gpio.h>
#include <linux/slab.h>                 //  kmalloc
#include <linux/uaccess.h>
#include <linux/semaphore.h>

#include "soc_AM335x.h"
#include "hw_cm_per.h"
#include "hw_gpmc.h"
#include "driver.h"

static void __iomem *soc_device_id;
struct semaphore mysem;

static struct file_operations sd_fops =
{
    write:              sd_write,
    read:               sd_read,
    open:               sd_open,
    release:            sd_release,
    unlocked_ioctl:     sd_ioctl,
};
static struct miscdevice sd_devs =
{
    minor:      MISC_DYNAMIC_MINOR,
    name:       DRIVER_NAME,
    fops:       &sd_fops
};

/*****************************************************************************
* @Function: pointers_init
* This function must provide initial values to all (u32 *) pointers used
* to address the  internal registers that must be configured in order
* to access the GPMC bus. Each assignation must be done using
* the function ioremap
 *****************************************************************************/
```

```
void pointers_init(void) {

	soc_device_id  =  ioremap(SOC_DEVICE_ID_REGS, 4);
}

/*****************************************************************************
 * @Function: gpmc_configuration
 *****************************************************************************/
uint gpmc_configuration(void) {

	return 0;
}

/*****************************************************************************
 * @Function: pin_mux_configuration
 *****************************************************************************/
int pin_mux_configuration(void) {

	return 0;
}

/*****************************************************************************
* @Function: gmpc_init
* This function will be called each time the module is uploaded.
* It must configure the GPMC registers (calling the gpmc_configuration()
* routine) and  it also has to adequately configure the pin multiplexation
* of the GPMC interface.
 *****************************************************************************/
int gpmc_init(void) {

	gpmc_configuration();
	pin_mux_configuration();

	return 0;
}

/*****************************************************************************
 * @Function: sd_open
 *****************************************************************************/
static int sd_open( struct inode* pInode, struct file* pFile ) {

	return 0;
}

/*****************************************************************************
 * @Function: sd_release
 *****************************************************************************/
static int sd_release( struct inode* pInode, struct file* pFile ) {

	return 0;
}

/*****************************************************************************
 * @Function: sd_ioctl
 *****************************************************************************/
static long sd_ioctl( struct file* pFile, unsigned int cmd, unsigned long value) {

	return 0;
}
```

```
/*****************************************************************************
 * @Function: sd_read
 *****************************************************************************/
static ssize_t sd_read(struct file *filp, char __user *buffer,
                size_t count, loff_t *f_pos) {

        u32 number;
        u32 *pointer=kzalloc(sizeof(u32), GFP_KERNEL);

        if (!pointer) {
                printk( KERN_WARNING DRIVER_NAME \
                        ": Problem allocating memory\n" );
                return(-1);
        }

        pointer[0] = hwreg(soc_device_id+DEVICE_FEATURE);
        number = copy_to_user(buffer,pointer,sizeof(u32));
        kfree(pointer);
        return number;
}

/*****************************************************************************
 * @Function: sd_write
 *****************************************************************************/
static ssize_t sd_write(struct file *filp, const char __user *buffer,
                size_t count, loff_t *f_pos) {

        return 0;
}

/*****************************************************************************
 * @Function: sd_init_module
 *****************************************************************************/
static int __init sd_init_module( void ) {

        pointers_init();
        gpmc_init();

        if( misc_register( &sd_devs ) )
        {
                printk( KERN_WARNING DRIVER_NAME \
                        ": The module cannot be registered\n" );
                return (-ENODEV);
        }

        printk( KERN_INFO "Module module.ko uploaded ******\n" );
        return 0;
}

/*****************************************************************************
 * @Function: sd_cleanup_module
 *****************************************************************************/
static void __exit sd_cleanup_module( void ) {

        misc_deregister( &sd_devs );

        printk( KERN_INFO "Module module.ko cleaned up ******\n" );
}
```

```
module_init( sd_init_module );
module_exit( sd_cleanup_module );

MODULE_LICENSE( "GPL" );
MODULE_AUTHOR("Digital Systems using Embedded Linux");
```

It is important to note that within a driver like this we have three different **memory address spaces**:

a) Physical addresses: the address values specified in the datasheet of the microcontroller. For example, the register with the identification of the microcontroller can be found at address 0x44E10600. There is a label defined in the *soc_AM335x.h* file for this specific address, SOC_DEVICE_ID_REGS:

```
static void_iomem *soc_device_id;    // Global variable

soc_device_id  =  ioremap(SOC_DEVICE_ID_REGS, 4);
```

The *ioremap* function generates a pointer in the virtual memory space of the kernel that will be mapped onto the physical address specified by the first argument (SOC_DEVICE_ID_REGS). The second argument is the memory size, in bytes, that must be allocated to the kernel. Since the identification register is 32 bit long, we only need 4 bytes in this case.

If, instead of this, we ever tried to execute:

```
aux = *(u32 *)0x44E10600;   // ERROR: not to be done like this
```

we would get a segmentation fault in kernel space, since the kernel must always work within the virtual address space that has been allocated to it. The address translation to the physical address space (the real one) is provided by the Memory Management unit (MMU) of the microprocessor.

Once the soc_device_id pointer has been initialized, we may access the contents of the ID register by simply executing:

```
        aux = *soc_device_id;
```

The previous instruction reads the contents of the identification register of the microprocessor and puts its contents into a variable called aux.

On the other hand, the following macro has been defined in the file driver.h:

```
        #define hwreg(x) *( volatile u32 *)((x))
```

The purpose of this definition is simply trying to make more evident that we are accessing an internal register of the microprocessor, not a typical kernel space address. If we want to use this macro for accessing the register we will execute:

```
        aux = hwreg(soc_device_id);
```

The registers may be accessed then in two different ways. You may choose whether to use the *hwreg* macro or not.

b) Kernel memory space. The MMU continually monitors whether all memory accesses inside the module have been adequately allocated to our module. This is a virtual memory space.

In order to allocate some memory to the kernel we may use different functions: `kmalloc, kzalloc, vmalloc` and others. In our example we use `kzalloc`:

```
u32 *pointer=kzalloc(sizeof(u32), GFP_KERNEL);
```

The first argument of this function is the size in bytes of the memory block we want to allocate. The second indicates the type of memory allocation we want to have. In this case the identifier `GFP_KERNEL` means that this is normal kernel allocation that can be lead to sleep (the process can be put to sleep while executing `kzalloc`). Finally, the difference between `kzalloc` and `kmalloc` is that the first one initializes the memory block to zero.

From the Linux `man` page we have:

```
DESCRIPTION

kmalloc is the normal method of allocating memory in the kernel.

The flags argument may be one of:

GFP_USER - Allocate memory on behalf of user. May sleep.
GFP_KERNEL - Allocate normal kernel ram. May sleep.
GFP_ATOMIC - Allocation will not sleep. Use inside interrupt handlers.
```

The `kfree` function deallocates the chunk of memory that has been previously allocated. It is important to allocate and deallocate correctly in order not to run out of memory.

c) User memory space: virtual memory allocated to the user. This memory cannot be directly addressed inside the module. This is important since the contents read out of the peripheral and given to the user application must be moved from the kernel space to the user space region:

```
unsigned long copy_to_user(void __user * to,
                           const void * from, unsigned long n);
```

The same happens with write operations; data from user applications must be passed along to the kernel using a special function:

```
unsigned long copy_from_user(void * to, const void __user * from,
                             unsigned long n);
```

FINAL NOTES:

1. the size (n) of the *copy_from_user* and *copy_to_user* routines is always specified in bytes.
2. The number of elements to be read or written specified in the *read* and *write* system calls (`size_t count`) is always expressed in bytes.
3. The *printk* function is the one that must be used within the kernel. In the Beaglebone, all kernel messages are directed towards the console (that can be accessed in the host PC at */dev/ttyUSB1* with the *screen* utility). These messages can also be read by executing the `dmesg` command.

An application using the driver *module.c* is the following (*app.c*):

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/types.h>
#include <sys/ioctl.h>

typedef unsigned short u16;

int main()
{
	unsigned long buffer[1];
	int f;

	if ((f=open("/dev/driver_port", O_RDWR))<0) {
		perror( "/dev/driver_port" );
		exit( EXIT_FAILURE );
	}

	if (read(f, buffer, sizeof(unsigned long))<0) {
		perror( "/dev/driver_port" );
		exit( EXIT_FAILURE );
	}
	printf("SDBL: Microprocessor feature reference AM3359 =  %lx\n"\,
		(unsigned long)buffer[0]);

	close(f);
	return EXIT_SUCCESS;
}
```

As a first step, the application opens the driver with the 'open' instruction. The driver must have been previously uploaded using the 'insmod' instruction. Once it has been opened, we may read the four bytes related to the identification register of the microprocessor.

1.3 Laboratory Work

Download the file `lm11.tar` from atenea and expand it in the directory `/beagle/driver` of the host PC. Unlike in previous Lab modules, now we are going to develop our programs in directories that will be mounted, so "seen" by the Beaglebone. It is highly recommended to backup these directories to a safe user area when finishing each Lab session.

1. In a host PC terminal, go to the driver directory and compile *module.c*:

```
ubuntu$ cd /beagle/driver/lm11
ubuntu$ make
```

The file `module.ko` is created in the directory `/beagle/driver/lm11/module`.

2. In a Beaglebone terminal, mount as usual the host PC directory and use the `lsmod` command to list the modules already loaded.

3. Use `insmod` to load the driver, as follows:

```
beaglebone#  portmap
beaglebone#  busybox mount -a
beaglebone#  cd /mnt/nfs_pc/driver/lm11/module
beaglebone#  insmod module.ko
```

Now you can use again `lsmod` to see that the new module has been added.

4. In the host PC terminal, compile *app.c*:

```
ubuntu$ cd /beagle/driver/lm11/application
ubuntu$ make
```

The executable file `app` is created in the directory `driver/lm11/application`.

5. Now we can execute the application in the Beaglebone:

```
beaglebone#  cd /mnt/nfs_pc/driver/lm11/application
beaglebone#  ./app
```

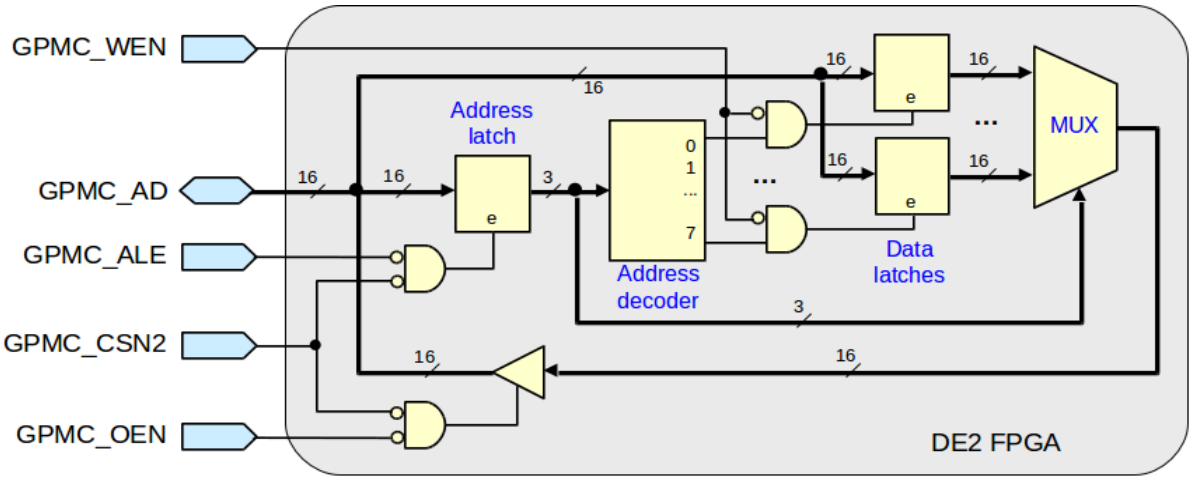
The steps explained here are the ones to follow when, in next sections, new driver versions and new applications will be developed.

2 GPMC Driver

The objective from now on is to configure a driver which implements a specific use of the General Purpose Memory Controller (GPMC) bus of the Beaglebone. We will use this driver to interchange data between the Beaglebone and the DE2 board.

2.1 The 'DE2 side'

Accordingly to the purpose explained above, the FPGA of the DE2 board has been programmed to work as an asynchronous memory device, available through the set of GPMC signals shown in the figure below.



The system implemented in the FPGA contains a bank composed of 8 data latches. By providing the appropriate asynchronous WRITE waveforms for Flash devices described later in this document, we can write a 16 bit data word from the GPMC_AD data/address bus to a given latch (specified by the 3 less significant bits of GPMC_AD, when it contains address data). It is also possible to read a 16-bit data word from any of the latches to GPMC_AD, using asynchronous READ waveforms.

Although not shown in the figure, the system implemented in the FPGA also allows to display the contents of the data latches, in the seven-segment displays available in the DE2 board.

2.2 The 'Beaglebone side'

2.2.1 GPMC Signals. NOR-mode multiplexed bus

The complete set of signals of the GPMC bus is the following:

7.1.4.1.2 Typical GPMC Setup

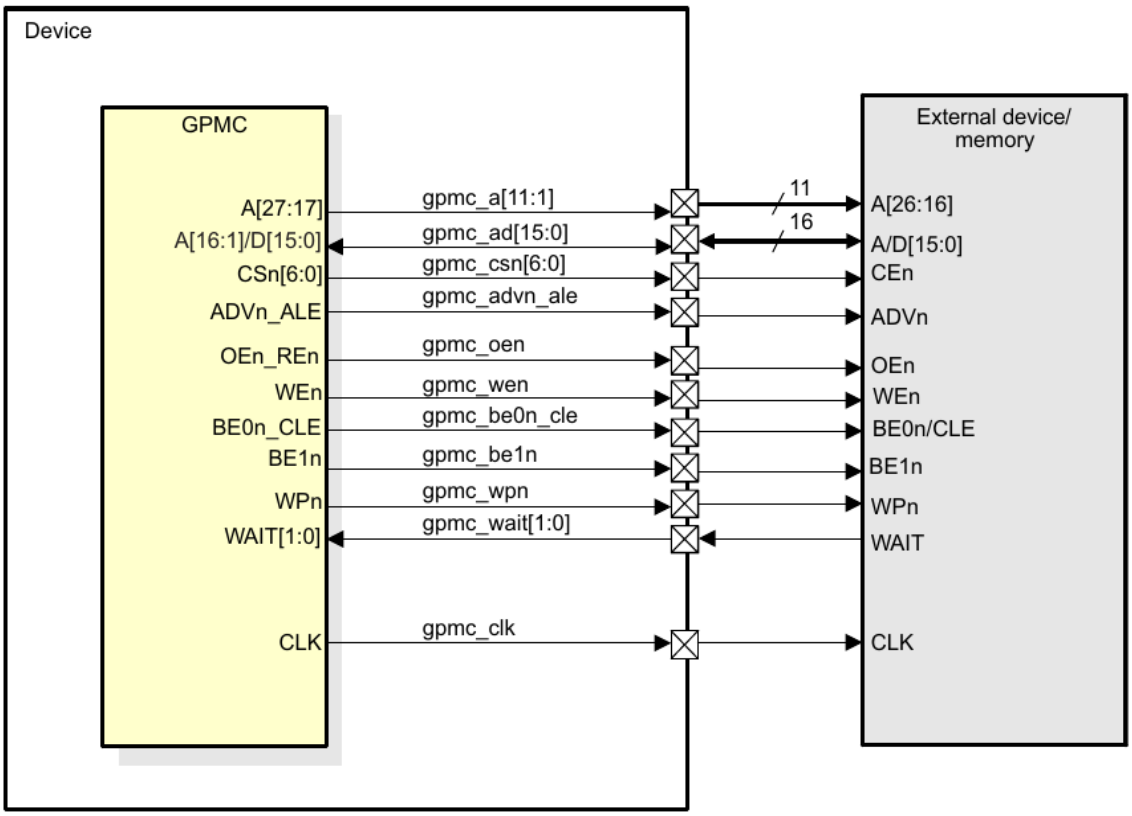
Table 7-45 lists some of the I/Os of the GPMC module.

Table 7-45. GPMC Signals

Signal Name	I/O	Description
GPMC_FCLK	Internal	Functional and interface clock. Acts as the time reference.
GPMC_CLK	O	External clock provided to the external device for synchronous operations
GPMC_A[27:17]	O	Address
GPMC_AD[15: 0]	I/O	Data-multiplexed with addresses A[16:1] on memory side
GPMC_CSxn	O	Chip-select (where x = 0, or 1)
GPMC_ADVn_ALE	O	Address valid enable
GPMC_OE_REn	O	Output enable (read access only)
GPMC_WEn	O	Write enable (write access only)
GPMC_WAIT[1:0]	I	Ready signal from memory device. Indicates when valid burst data is ready to be read

A typical connection with a memory can be done in the following way:

Figure 7-3. GPMC to 16-Bit Address/Data-Multiplexed Memory

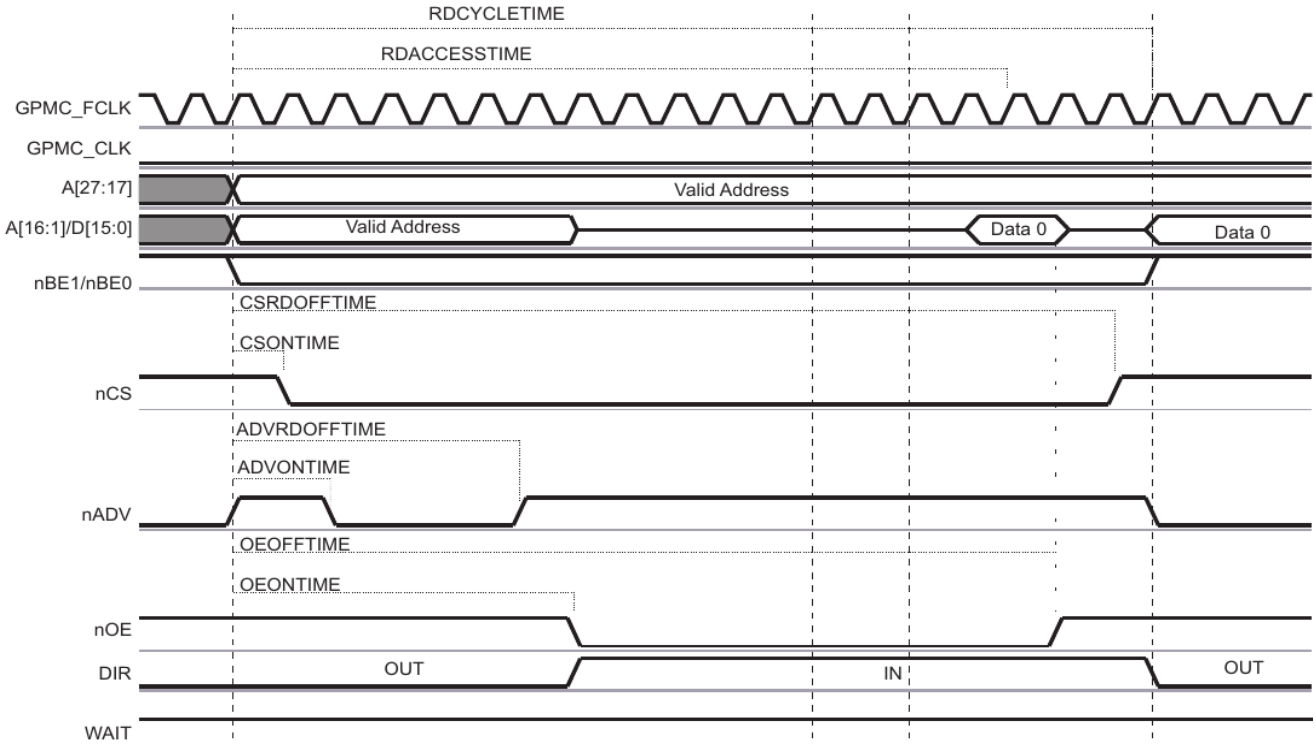


2.2.2 GPMC Timing

The main parameters governing GPMC timing for Read and Write operations on Address/Data Multiplexed devices are:

For READ operations:

Figure 7-12. Asynchronous Single Read Operation on an Address/Data Multiplexed Device



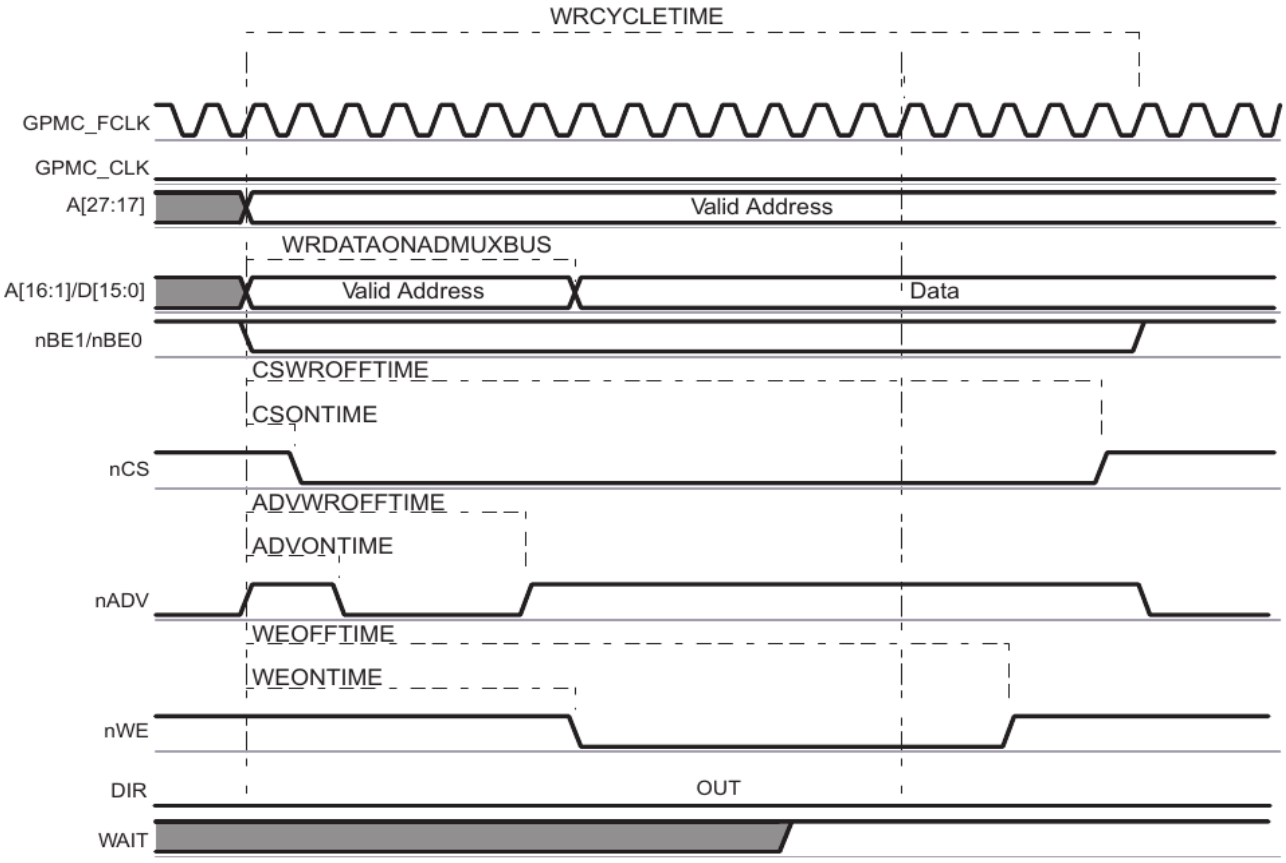
Therefore we have to program the following parameters:

- RDCYCLETIME
- RDACCESSTIME
- CSONTIME
- CSRDOFFTIME
- CSEXTRADELAY (delay the CS# signal for half a clock period)
- ADVONTIME
- ADVRDOFFTIME
- OEONTIME
- OEOFFTIME

Note that when the nOE signal is de-asserted (OEOFFTIME) the Address in the multiplexed bus disappears and the bus changes to INPUT MODE (check the value of the DIR pin).

For WRITE operations:

Figure 7-14. Asynchronous Single Write on an Address/Data-Multiplexed Device



Now, for write operations we have to program the following parameters:

- WRCYCLETIME
- CSONTIME: (the same value as for read operations)
- CSWROFFTIME
- ADVONTIME (the same value as for read operations)
- ADVWROFFTIME
- WEONTIME
- WEOFFTIME
- WRDATAONADMUXBUS

The next page lists ALL the parameters that can be programmed as a function of the access type.

Table 7-41. Timing Parameters

Register	Bit	Bit Field Name	Asynchronous			Synchronous				Access Type		
			Single Read Access	Single Write Access	Multiple Read (Page) access	Single Read Access	Single Write Access	Multiple Read (Burst) Access	Multiple Write (Burst) Access	Non-multiplexed	Address /Data-multiplexed	AAD-multiplexed
GPMC_CONFIG1_i	9-8	MUXADDDATA	y	y	y	y	y	y	y	y	y	y
GPMC_CONFIG1_i	29	READTYPE	y		y	y		y		y	y	y
GPMC_CONFIG1_i	30	READMULTIPLE	y		y	y		y		y	y	y
GPMC_CONFIG1_i	27	WRITETYPE		y			y		y	y	y	y
GPMC_CONFIG1_i	28	WRITEMULTIPLE		y			y		y	y	y	y
GPMC_CONFIG1_i	31	WRAPBURST						y	y	y	y	y
GPMC_CONFIG1_i	26-25	CLKACTIVATIONTIME				y	y	y	y	y	y	y
GPMC_CONFIG1_i	19-18	WAITMONITORINGTIME	y	y	y	y	y	y	y	y	y	y
GPMC_CONFIG1_i	4	TIMEPARAGRANULARITY	y	y	y	y	y	y	y	y	y	y
GPMC_CONFIG2_i	20-16	CSWROFFTIME		y			y		y	y	y	y
GPMC_CONFIG2_i	12-8	CSRDOFFTIME	y		y	y		y		y	y	y
GPMC_CONFIG2_i	7	CSEXTRADELAY	y	y	y	y	y	y	y	y	y	y
GPMC_CONFIG2_i	3-0	CSONTIME	y	y	y	y	y	y	y	y	y	y
GPMC_CONFIG3_i	30-28	ADVAADMUXWROFFTIME		y			y		y			y
GPMC_CONFIG3_i	26-24	ADVAADMUXRDOFFTIME	y		y	y		y				y
GPMC_CONFIG3_i	6-4	ADVAADMUXONTIME	y	y	y	y	y	y	y			y
GPMC_CONFIG3_i	20-16	ADVWROFFTIME		y			y		y	y	y	y
GPMC_CONFIG3_i	12-8	ADVRDOFFTIME	y		y	y		y		y	y	y
GPMC_CONFIG3_i	7	ADVEXTRADELAY	y	y	y	y	y	y	y	y	y	y
GPMC_CONFIG3_i	3-0	ADVONTIME	y	y	y	y	y	y	y	y	y	y
GPMC_CONFIG4_i	15-13	OEAADMUXOFFTIME	y	y	y	y	y	y	y			y
GPMC_CONFIG4_i	6-4	OEAADMUXONTIME	y	y	y	y	y	y	y			y
GPMC_CONFIG4_i	28-24	WEOFFTIME		y			y		y	y	y	y
GPMC_CONFIG4_i	23	WEEXTRADELAY		y			y		y	y	y	y
GPMC_CONFIG4_i	19-16	WEONTIME		y			y		y	y	y	y
GPMC_CONFIG4_i	12-8	OEOFFTIME	y		y	y		y		y	y	y
GPMC_CONFIG4_i	7	OEEXTRADELAY	y		y	y		y		y	y	y
GPMC_CONFIG4_i	3-0	OEONTIME	y		y	y		y		y	y	y
GPMC_CONFIG5_i	27-24	PAGEBURSTACCESSTIME			y			y	y	y	y	y
GPMC_CONFIG5_i	20-16	RDACCESSTIME	y		y	y		y		y	y	y
GPMC_CONFIG5_i	12-8	WRCYCLETIME		y			y		y	y	y	y
GPMC_CONFIG5_i	4-0	RDCYCLETIME	y		y	y		y		y	y	y
GPMC_CONFIG6_i	28-24	WRACCESSTIME		y			y		y	y	y	y
GPMC_CONFIG6_i	19-16	WRDATAONADMUXBUS		y			y		y		y	y
GPMC_CONFIG6_i	11-8	CYCLE2CYCLEDELAY	y	y	y	y	y	y	y	y	y	y
GPMC_CONFIG6_i	7	CYCLE2CYCLESAMECSEN	y	y	y	y	y	y	y	y	y	y
GPMC_CONFIG6_i	6	CYCLE2CYCLEDIFFCSEN	y	y	y	y	y	y	y	y	y	y
GPMC_CONFIG6_i	3-0	BUSTURNAROUND	y	y	y	y	y	y	y	y	y	y
GPMC_CONFIG7_i	6	CSVALID	y	y	y	y	y	y	y	y	y	y

2.2.3 Example: Asynchronous Flash NOR device

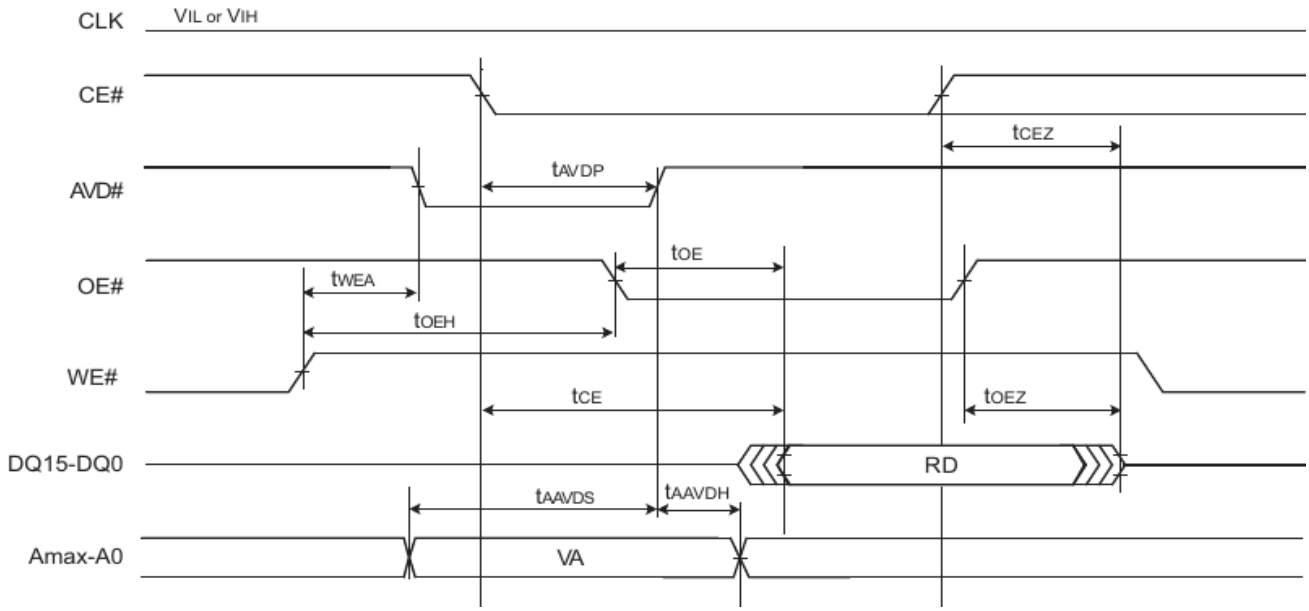
Now, we will proceed with the example provided in the technical reference of the microcontroller. First it provides the access time parameters of a given memory and then it gives a set of possible values for the timing parameters of the GPMC bus. Although we won't be using these parameters in this laboratory, this is a good example of how to provide good values to the numerous timing parameters of the GPMC bus.

READ CYCLE:

Table 7-48. AC Characteristics for Asynchronous Read Access

AC Read Characteristics on the Memory Side	Description	Duration (ns)
tCE	Read Access time from CSn low	80 (max)
tAAVDS	Address setup time to rising edge of ADVn	3 (min)
tAVDP	ADVn low time	6 (min)
tCAS	CSn setup time to ADVn	0 (min)
tOE	Output enable to output valid	6 (max)
tOEZ	Output enable to High-Impedance	7 (max)

Figure 11.9 Asynchronous Read Mode (AVD# Toggling - Case 1)



For this case, since t_{CAS}(min)=0, we may activate CE# after activation of AVD#.

Now, from the point of view of the GPMC we have that the following parameters of the GPMC access can be programmed as follows:

Use the following formula to calculate the RdCycleTime parameter for this typical access:

RdCycleTime = RdAccessTime + AccessCompletion = RdAccessTime + 1 clock cycle + tOEZ

- First, on the memory side, the external memory makes the data available to the output bus. This is the memory-side read access time defined in Table 7-49: the number of clock cycles between the address capture (ADVn rising edge) and the data valid on the output bus.

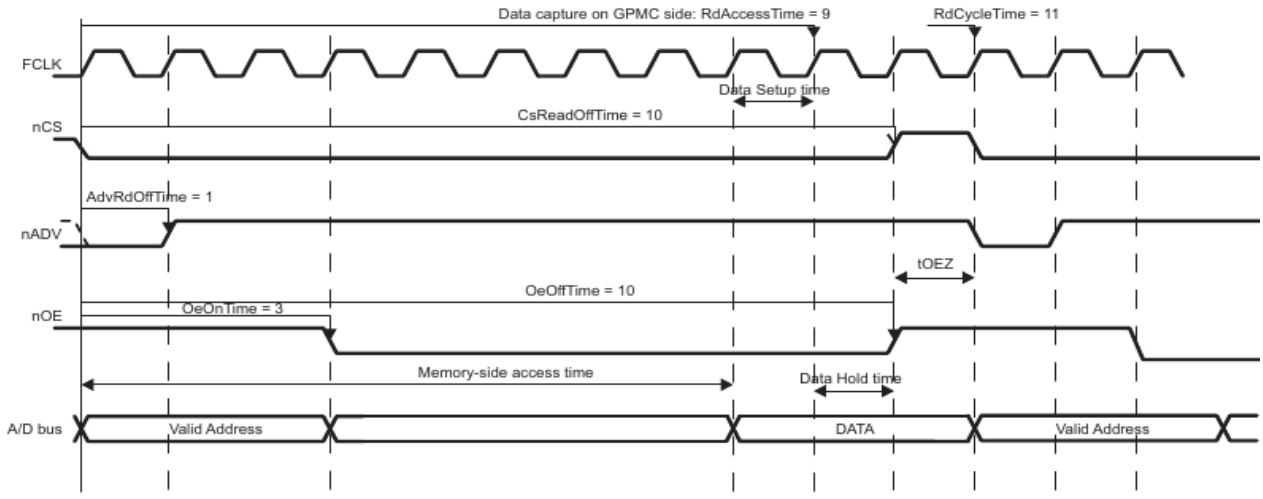
The GPMC requires some hold time to allow the data to be captured correctly and the access to be finished.

- To read the data correctly, the GPMC must be configure to meet the data setup time requirement of the memory; the GPMC module captures the data on the next rising edge. This is access time on the GPMC side.
- There must also be a data hold time for correctly reading the data (checking that there is no OEn/CSn deassertion while reading the data). This data hold time is 1 clock cycle (that is, AccessTime + 1).
- To complete the access, OEn/CSn signals are driven to high-impedance. AccessTime + 1 + tOEZ is the read cycle time.
- Addresses can now be relatched and a new read cycle begun.

Table 7-49. GPMC Timing Parameters for Asynchronous Read Access

Parameter Name on GPMC side	Formula	Duration (ns)	Number of Clock Cycles (F = 104 MHz)	GPMC Register Configurations
ClkActivationTime	n/a (asynchronous mode)			
AccessTime	round max (tCE)	80	9	ACCESSTIME = 9h
PageBurstAccessTime	n/a (single access)			
RdCycleTime	AccessTime + 1cycle + tOEZ	96, 615	11	RDCYCLETIME = Bh
CsOnTime	tCAS	0	0	CSONTIME = 0
CsReadOffTime	AccessTime + 1 cycle	89, 615	10	CSRDOFFTIME = Ah
AdvOnTime	tAAVDS	3	1	ADVONTIME = 1
AdvRdOffTime	tAAVDS + tAVDP	9	1	ADVRDOFFTIME = 1
OeOnTime	OeOnTime ≥ AdvRdOffTime (multiplexed mode)	-	3, for instance	OEONTIME = 3h
OeOffTime	AccessTime + 1cycle	89, 615	10	OEOFFTIME = Ah

Figure 7-49. Asynchronous Single Read Access (Timing Parameters in Clock Cycles)



2.3 Programming the GPMC driver for the DE2

2.3.1 Enable clock for GPMC module:

a) As a first step we will enable the clock for the GPMC module by writing the MODULEMODE field of the CM_PER_GPMC_CLKCTRL register.

b) In a second step we will check that it is indeed enabled: we will wait till the IDLEST bits (bits 17-16) take the value '00' (corresponding to “Func”, so that the module is fully functional).

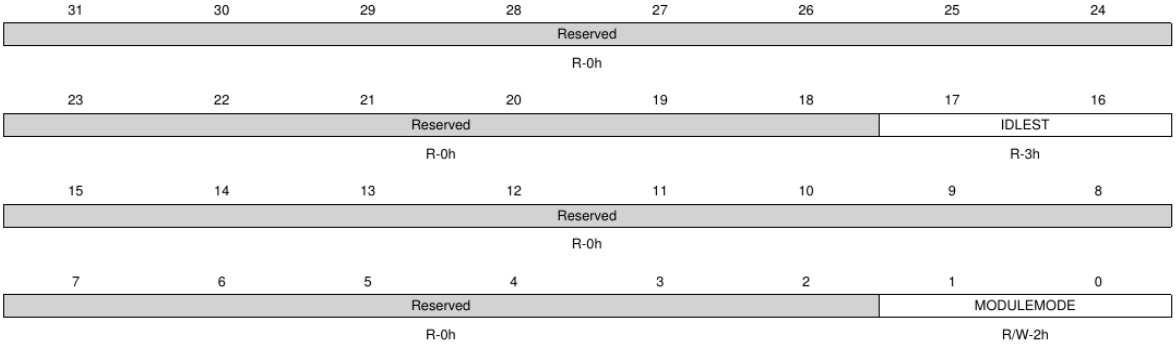
The offset of the CM_PER_GPMC_CLKCTRL register, =0x30, defined in hw_cm_per.h, is specified from the base address given by SOC_PRCM_REGS, given in the file soc_AM335.h.

8.1.12.1.11 CM_PER_GPMC_CLKCTRL Register (offset = 30h) [reset = 30002h]

CM_PER_GPMC_CLKCTRL is shown in Figure 8-33 and described in Table 8-40.

This register manages the GPMC clocks.

Figure 8-33. CM_PER_GPMC_CLKCTRL Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -*n* = value after reset

Table 8-40. CM_PER_GPMC_CLKCTRL Register Field Descriptions

Bit	Field	Type	Reset	Description
31-18	Reserved	R	0h	
17-16	IDLEST	R	3h	Module idle status. 0x0 = Func : Module is fully functional, including OCP 0x1 = Trans : Module is performing transition: wakeup, or sleep, or sleep abortion 0x2 = Idle : Module is in Idle mode (only OCP part). It is functional if using separate functional clock 0x3 = Disable : Module is disabled and cannot be accessed
15-2	Reserved	R	0h	
1-0	MODULEMODE	R/W	2h	Control the way mandatory clocks are managed. 0x0 = DISABLED : Module is disable by SW. Any OCP access to module results in an error, except if resulting from a module wakeup (asynchronous wakeup). 0x1 = RESERVED_1 : Reserved 0x2 = ENABLE : Module is explicitly enabled. Interface clock (if not used for functions) may be gated according to the clock domain state. Functional clocks are guarantied to stay present. As long as in this configuration, power domain sleep transition cannot happen. 0x3 = RESERVED : Reserved

2.3.2 Reset the GPMC module:

a) We will reset the GPMC module (a software reset since this is done once the system has been initialized). To this effect we will activate the SOFTRESET bit in the GPMC_SYSCONFIG register (without changing the contents of all other bits, i.e. using masks).

The offset of the GPMC_SYSCONFIG register, =0x10, is specified from the base address given by SOC_GPMC_0_REGS, given in the file soc_AM335.h.

7.1.5.2 GPMC_SYSCONFIG

This register controls the various parameters of the OCP interface.

Figure 7-52. GPMC_SYSCONFIG

31							16
Reserved							
R-0							
15		5	4	3	2	1	0
Reserved			SIDLEMODE		Rsvd	SOFTRESET	AUTOIDLE
R-0			R/W		R-0	R/W	R/W

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

Table 7-56. GPMC_SYSCONFIG Field Descriptions

Bit	Field	Value	Description
31-5	Reserved	0	Reserved
4-3	SIDLEMODE		Idle mode
		0	Force-idle. An idle request is acknowledged unconditionally
		1h	No-idle. An idle request is never acknowledged
		2h	Smart-idle. Acknowledgement to an idle request is given based on the internal activity of the module
		3h	Reserved
2	Reserved	0	Reserved
1	SOFTRESET		Software reset (Set 1 to this bit triggers a module reset. This bit is automatically reset by hardware. During reads, it always returns 0)
		0	Normal mode
		1	The module is reset
0	AUTOIDLE		Internal OCP clock gating strategy
		0	Interface clock is free-running
		1	Reserved

b) Now, as a second step we will check that the reset has been completed: to this effect we will check the GPMC_SYSSTATUS register.

We will wait till the bit RESETDONE is equal to 1.

The offset of the GPMC_SYSSTATUS register, =0x14, defined in hw_gpmc.h, is specified from the base address given by SOC_GPMC_0_REGS, given in the file soc_AM335.h.

7.1.5.3 GPMC_SYSSTATUS

This register provides status information about the module, excluding the interrupt status information.

Figure 7-53. GPMC_SYSSTATUS

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
R-0															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															RESETDONE
R-0															R-0

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

Table 7-57. GPMC_SYSSTATUS Field Descriptions

Bit	Field	Value	Description
31-1	Reserved	0	Reserved
0	RESETDONE		Internal reset monitoring
		R0	Internal module reset in on-going
		R1	Reset completed

2.3.3 Configure GPMC to NO IDLE

We will configure the SIDLEMODE bits (bits 4-3) of the GPMC_SYSCONFIG register to 0x1: 'No idle. An idle request is never acknowledged'.

Use a temporary variable to do any intermediate operation so that you only write once into the GPMC_SYSCONFIG register. The reason for doing this is to avoid going through an intermediate configuration that sometimes may generate problems.

For example, while writing the SIDLEMODE bits to 01 we might be using a mask so that first we put both bits to '00' and then, in a second operation, we switch to '01'. If we do this we will configuring the GPMC to be idle for some time, and we don't know the repercussions this may have.

2.3.4 Mask all interrupts from the GPMC module

We will mask all interrupts of the GPMC module by configuring the GPMC_IRQENABLE to 0x00. This register has an offset = 0x1C, defined in hw_gpmc.h, from the base address SOC_GPMC_0_REGS, given in the file soc_AM335.h

7.1.5.5 GPMC_IRQENABLE

The interrupt enable register allows to mask/unmask the module internal sources of interrupt, on a event-by-event basis.

Figure 7-55. GPMC_IRQENABLE

31	16
Reserved	
R-0	

15	10	9	8
Reserved		WAIT1EDGE DETECTIONENABLE	WAIT0EDGE DETECTIONENABLE
R-0		R/W-0	R/W-0

7	2	1	0
Reserved		TERMINAL COUNTENABLE	FIFOEVENT ENABLE
R-0		R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

Table 7-59. GPMC_IRQENABLE Field Descriptions

Bit	Field	Value	Description
31-10	Reserved	0	Reserved
9	WAIT1EDGEDETECTIONENABLE		Enables the Wait1 Edge Detection interrupt
		0	Wait1EdgeDetection interrupt is masked
		1	Wait1EdgeDetection event generates an interrupt if occurs
8	WAIT0EDGEDETECTIONENABLE		Enables the Wait0 Edge Detection interrupt
		0	Wait0EdgeDetection interrupt is masked
		1	Wait0EdgeDetection event generates an interrupt if occurs
7-2	Reserved	0	Reserved
1	TERMINALCOUNTEVENTENABLE		Enables TerminalCountEvent interrupt issuing in pre-fetch or write posting mode
		0	TerminalCountEvent interrupt is masked
		1	TerminalCountEvent interrupt is not masked
0	FIFOEVENTENABLE		Enables the FIFOEvent interrupt
		0	FIFOEvent interrupt is masked
		1	FIFOEvent interrupt is not masked

2.3.5 Disable TIMEOUT_CONTROL

Since the device we will be connecting to the GPMC bus is very fast we won't be using the TIMEOUT_CONTROL feature of the AM3359 processor. From the reference manual we have:

- ERRORTIMEOUT: A time-out mechanism prevents the system from hanging. The start value of the

9-bit time-out counter is defined in the GPMC_TIMEOUT_CONTROL register and enabled with the GPMC_TIMEOUT_CONTROL[0] TIMEOUTENABLE bit. When enabled, the counter starts at start-cycle time until it reaches 0 and data is not responded to from memory, and then a time-out error occurs. When data are sent from memory, this counter is reset to its start value. With multiple accesses (asynchronous page mode or synchronous burst mode), the counter is reset to its start value for each data access within the burst.

We will therefore write a 0x00 in the GPMC_TIMEOUT_CONTROL register.

This register has an offset = 0x1C, defined in hw_gpmc.h, from the base address SOC_GPMC_0_REGS, given in the file soc_AM335.h.

7.1.5.6 GPMC_TIMEOUT_CONTROL

The GPMC_TIMEOUT_CONTROL register allows the user to set the start value of the timeout counter

Figure 7-56. GPMC_TIMEOUT_CONTROL

31	16
Reserved	
R-0	

15	13	12	4	3	1	0
Reserved		TIMEOUTSTARTVALUE		Reserved		TIMEOUTENABLE
R-0		R/W-0		R-0		R/W-0

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

Table 7-60. GPMC_TIMEOUT_CONTROL Field Descriptions

Bit	Field	Value	Description
31-13	Reserved	0	Reserved
12-4	TIMEOUTSTARTVALUE	0-1FFh	Start value of the time-out counter (000 corresponds to 0 GPMC.FCLK cycle, 1h corresponds to 1 GPMC.FCLK cycle, and 1FFh corresponds to 511 GPMC.FCLK cycles)
3-1	Reserved	0	Reserved
0	TIMEOUTENABLE		Enable bit of the TimeOut feature
		0	TimeOut feature is disabled
		1	TimeOut feature is enabled

2.3.6 TIMING configuration for the GPMC bus

Now, we will be configuring in 7 steps the GPMC configuration for a given chip select. To this effect there are 7 configuration registers for each chip select.

In order to clarify programs we will be using systematically the macros made by Texas Instruments and defined in *hw_gpmc.h*.

```
#define  GPMC_CONFIG1(n)  (0x60 + (n * (0x30)))
#define  GPMC_CONFIG2(n)  (0x64 + (n * (0x30)))
#define  GPMC_CONFIG3(n)  (0x68 + (n * (0x30)))
#define  GPMC_CONFIG4(n)  (0x6C + (n * (0x30)))
#define  GPMC_CONFIG5(n)  (0x70 + (n * (0x30)))
#define  GPMC_CONFIG6(n)  (0x74 + (n * (0x30)))
#define  GPMC_CONFIG7(n)  (0x78 + (n * (0x30)))
```

These macros provide the offset of the corresponding GPMC_CONFIGx register depending on number of chip select we will be using.

This means that in order to write a particular value to the the GPMC_CONFIG1 register, for example, related to the chip select *csNum* we will be using:

```
static void __iomem *soc_gpmc_base;

// base pointer preparation
soc_gpmc_base = ioremap(SOC_GPMC_0_REGS, SZ_16M);

// Access to GPMC_CONFIG1_i, with i=csNum
hwreg(soc_gpmc_base + GPMC_CONFIG1(csNum)) = XXX;
```

2.3.7 GPMC_CONFIG1

Let us now configure the GPMC bus for NOR accesses with x2 granularity and multiplexed bus AD0-AD15.

First we will program the following content of the configuration bits:

DEVICESIZE (bits 13-12) = 1h	(16 bit device)
ATTACHEDDEVICEPAGELENGTH (bits 24-23)=0	(burst length of the attached device = 4 words)
MUXADDDATA (bits 9-8) = 0x2	(Address and Data multiplexed)

All other bits can be set to 0. In particular note that from the DEVICETYPE field this means that we will be accessing a NOR device, with asynchronous reads and writes, in single mode (no burst), without monitoring of the WAIT signal and without dividing the GPMC_FCLOCK frequency.

We will be using TIMEPARAGRANULARITY=0. This is an important parameter that allows scaling all the times we will be defining from here onwards.

7.1.5.11 GPMC_CONFIG1_i

The configuration 1 register sets signal control parameters per chip select.

Figure 7-61. GPMC_CONFIG1_i

31	30	29	28	27	26	25	24
WRAPBURST	READMULTIPLE	READTYPE	WRITEMULTIPLE	WRITETYPE	CLKACTIVATIONTIME		ATTACHEDDEVICE PAGELENGTH
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0		R/W-0

23	22	21	20	19	18	17	16
ATTACHEDDEVICE PAGELENGTH	WAITREADMONITORING	WAITWRITEMONITORING	Reserved	WAITMONITORINGTIME		WAITPINSELECT	
R/W-0	R/W-0	R/W-0	R-0	R/W-0		R/W-0	

15	14	13	12	11	10	9	8
Reserved		DEVICESIZE		DEVICETYPE		MUXADDDATA	
R-0		R/W-0		R/W-0		R/W-0	

7	5	4	3	2	1	0
Reserved		TIMEPARA GRANULARITY	Reserved		GPMCFCLKDIVIDER	
R-0		R/W-0	R-0		R/W-0	

LEGEND: R = Read only; W1C = Write 1 to clear bit; -*n* = value after reset

Table 7-65. GPMC_CONFIG1_i Field Descriptions

Bit	Field	Value	Description
31	WRAPBURST		Enables the wrapping burst capability. Must be set if the attached device is configured in wrapping burst
		0	Synchronous wrapping burst not supported
		1	Synchronous wrapping burst supported
30	READMULTIPLE		Selects the read single or multiple access
		0	single access
		1	multiple access (burst if synchonous, page if asynchronous)
29	READTYPE		Selects the read mode operation
		0	Read Asynchronous
		1	Read Synchronous
28	WRITEMULTIPLE		Selects the write single or multiple access
		0	Single access
		1	Multiple access (burst if synchronous, considered as single if asynchronous)
27	WRITETYPE		Selects the write mode operation
		0	Write Asynchronous
		1	Write Synchronous
26-25	CLKACTIVATIONTIME		Output GPMC.CLK activation time
		0	First rising edge of GPMC_CLK at start access time
		1h	First rising edge of GPMC_CLK one GPMC_FCLK cycle after start access time
		2h	First rising edge of GPMC_CLK two GPMC_FCLK cycles after start access time
		3h	Reserved
24-23	ATTACHEDDEVICEPAGELENGTH		Specifies the attached device page (burst) length (1 Word = Interface size)
		0	4 Words
		1h	8 Words
		2h	16 Words
		3h	Reserved

Table 7-65. GPMC_CONFIG1_i Field Descriptions (continued)

Bit	Field	Value	Description
22	WAITREADMONITORING		Selects the Wait monitoring configuration for Read accesses
		0	WAIT pin is not monitored for read accesses
		1	WAIT pin is monitored for read accesses
21	WAITWRITEMONITORING		Selects the Wait monitoring configuration for Write accesses
		0	WAIT pin is not monitored for write accesses
		1	WAIT pin is monitored for write accesses
20	Reserved	0	Reserved
19-18	WAITMONITORINGTIME		Selects input pin Wait monitoring time
		0	WAIT pin is monitored with valid data
		1h	WAIT pin is monitored one GPMC_CLK cycle before valid data
		2h	WAIt pin is monitored two GPMC_CLK cycle before valid data
		3h	Reserved
17-16	WAITPINSELECT		Selects the input WAIT pin for this chip select
		0	WAIT input pin is WAIT0
		1h	WAIT input pin is WAIT1
		2h	Reserved
		3h	Reserved
15-14	Reserved	0	Reserved
13-12	DEVICESIZE		Selects the device size attached.
		0	8 bit
		1h	16 bit
		2h	Reserved
		3h	Reserved
11-10	DEVICETYPE		Selects the attached device type
		0	NOR Flash like, asynchronous and synchronous devices
		1h	Reserved
		2h	NAND Flash like devices, stream mode
		3h	Reserved
9-8	MUXADDDATA		Enables the Address and data multiplexed protocol (Reset value is SYSBOOT[11:10] input pin sampled at IC reset for $\overline{CS}$[0] and 0 for $\overline{CS}$[1-6])
		0	Non-multiplexed attached device
		1h	AAD-multiplexed protocol device
		2h	Address and data multiplexed attached device
		3h	Reserved
7-5	Reserved	0	Reserved
4	TIMEPARAGRANULARITY		Signals timing latencies scalar factor (Rd/WRCycleTime, AccessTime, PageBurstAccessTime, CSOnTime, CSRd/WrOffTime, ADVOnTime, ADVRd/WrOffTime, OEOnTime, OEOffTime, WEOnTime, WEOffTime, Cycle2CycleDelay, BusTurnAround, TimeOutStartValue)
		0	×1 latencies
		1	×2 latencies
3-2	Reserved	0	Reserved
1-0	GPMCFCLKDIVIDER		Divides the GPMC.FCLK clock
		0	GPMC_CLK frequency = GPMC_FCLK frequency
		1h	GPMC_CLK frequency = GPMC_FCLK frequency/2
		2h	GPMC_CLK frequency = GPMC_FCLK frequency/3
		3h	GPMC_CLK frequency = GPMC_FCLK frequency/4

2.3.8 GPMC_CONFIG2

Now, we will configure the timing of the CS# signal for read and write cycles.

7.1.5.12 GPMC_CONFIG2_i

Chip-select signal timing parameter configuration.

Figure 7-62. GPMC_CONFIG2_i

31	21	20	16
Reserved		CSWROFFTIME	
R-0		R/W-0	

15	13	12	8	7	6	4	3	0
Reserved		CSRDOFFTIME		CSEXTRADELAY	Reserved		CSONTIME	
R-0		R/W-0		R/W-0	R-0		R/W-0	

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

Table 7-66. GPMC_CONFIG2_i Field Descriptions

Bit	Field	Value	Description
31-21	Reserved	0	Reserved
20-16	CSWROFFTIME		CS# de-assertion time from start cycle time for write accesses
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		1Fh	31 GPMC_FCLK cycles
15-13	Reserved	0	Reserved
12-8	CSRDOFFTIME		CS# de-assertion time from start cycle time for read accesses
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		1Fh	31 GPMC_FCLK cycles
7	CSEXTRADELAY		CS# Add Extra Half GPMC.FCLK cycle
		0	CS i Timing control signal is not delayed
		1	CS i Timing control signal is delayed of half GPMC_FCLK clock cycle
6-4	Reserved	0	Reserved
3-0	CSONTIME		CS# assertion time from start cycle time
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		1Fh	15 GPMC_FCLK cycles

We will program:

CSONTIME	=	0x01
CSRDOFFTIME	=	0xa
CSWROFFTIME	=	0xa
CSEXTRADELAY	=	0x0

2.3.9 GPMC_CONFIG3

Now, we configure ADVONTIME, ADVRDOFFTIME and ADVWROFFTIME

7.1.5.13 GPMC_CONFIG3_i

ADV# signal timing parameter configuration.

Figure 7-63. GPMC_CONFIG3_i

31	30	28	27	26	24
Reserved	ADVAADMUXWROFFTIME		Reserved	ADVAADMUXRDOFFTIME	
R-0	R/W-0		R-0	R/W-0	

23	21	20	16
Reserved		ADVWROFFTIME	
R/W-0		R/W-0	

15	13	12	8
Reserved		ADVRDOFFTIME	
R-0		R/W-0	

7	6	4	3	0
ADVEXTRA DELAY	ADVAADMUXONTIME		ADVONTIME	
R/W-0	R/W-0		R/W-0	

LEGEND: R = Read only; W1C = Write 1 to clear bit; -*n* = value after reset

Table 7-67. GPMC_CONFIG3_i Field Descriptions

Bit	Field	Value	Description
31	Reserved	0	Reserved
30-28	ADVAADMUXWROFFTIME		ADV# de-assertion for first address phase when using the AAD-Mux protocol
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		7h	7 GPMC_FCLK cycles
27	Reserved	0	Reserved
26-24	ADVAADMUXRDOFFTIME		ADV# assertion for first address phase when using the AAD-Mux protocol
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		7h	7 GPMC_FCLK cycles
23-21	Reserved	0	Reserved
20-16	ADVWROFFTIME		ADV# de-assertion time from start cycle time for write accesses
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		1Fh	31 GPMC_FCLK cycles
15-13	Reserved	0	Reserved
12-8	ADVRDOFFTIME		ADV# de-assertion time from start cycle time for read accesses
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		1Fh	31 GPMC_FCLK cycles
7	ADVEXTRADELAY		ADV# Add Extra Half GPMC.FCLK cycle
		0	$\overline{\text{ADV}}$ Timing control signal is not delayed
		1	$\overline{\text{ADV}}$ Timing control signal is delayed of half GPMC_FCLK clock cycle

Table 7-67. GPMC_CONFIG3_i Field Descriptions (continued)

Bit	Field	Value	Description
6-4	ADVAADMUXONTIME		ADV# assertion for first address phase when using the AAD-Multiplexed protocol
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		7h	7 GPMC_FCLK cycles
3-0	ADVONTIME		ADV# assertion time from start cycle time
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		Fh	15 GPMC_FCLK cycles

Possible values for these parameters are:

ADVONTIME = 1

ADVRDOFFTIME = 3

ADVWROFFTIME = 3

All other parameters can be set to 0.

2.3.10 GPMC_CONFIG4

In this section we will configure the OEONTIME and OEOFFTIME (assertion and de-assertion times during the read cycle for OE*), together with WEONTIME and WEOFFTIME (idem for WE* during write cycles).

7.1.5.14 GPMC_CONFIG4_i

WE# and OE# signals timing parameter configuration.

Figure 7-64. GPMC_CONFIG4_i

31	29	28	24
Reserved		WEOFFTIME	
R-0		R/W-0	

23	22	20	19	16
WEEXTRADELAY	Reserved		WEONTIME	
R/W-0	R-0		R/W-0	

15	13	12	8
OEAADMUXOFFTIME		OEOFFTIME	
R/W-0		R/W-0	

7	6	4	3	0
OEEXTRA DELAY	OEAADMUXONTIME		OEONTIME	
R/W-0	R/W-0		R/W-0	

LEGEND: R = Read only; W1C = Write 1 to clear bit; -*n* = value after reset

Table 7-68. GPMC_CONFIG4_i Field Descriptions

Bit	Field	Value	Description
31-29	Reserved	0	Reserved
28-24	WEOFFTIME		WE# de-assertion time from start cycle time
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		1Fh	31 GPMC_FCLK cycles
23	WEEXTRADELAY		WE# Add Extra Half GPMC.FCLK cycle
		0	$\overline{WE}$ Timing control signal is not delayed
		1	$\overline{WE}$ Timing control signal is delayed of half GPMC_FCLK clock cycle
22-20	Reserved	0	Reserved
19-16	WEONTIME		WE# assertion time from start cycle time
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		Fh	15 GPMC_FCLK cycles
15-13	OEAADMUXOFFTIME		OE# de-assertion time for the first address phase in an AAD-Multiplexed access
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		7h	7 GPMC_FCLK cycles
12-8	OEOFFTIME		OE# de-assertion time from start cycle time
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		1Fh	31 GPMC_FCLK cycles
7	OEEXTRADELAY		OE# Add Extra Half GPMC.FCLK cycle
		0	$\overline{OE}$ Timing control signal is not delayed
		1	$\overline{OE}$ Timing control signal is delayed of half GPMC_FCLK clock cycle

Table 7-68. GPMC_CONFIG4_i Field Descriptions (continued)

Bit	Field	Value	Description
6-4	OEAADMUXONTIME		OE# assertion time for the first address phase in an AAD-Multiplexed access
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		7h	7 GPMC_FCLK cycles
3-0	OEONTIME		OE# assertion time from start cycle time
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		Fh	15 GPMC_FCLK cycles

Possible values for these parameters are:

OEONTIME = 5
OEOFFTIME = 9
WEONTIME = 5
WEOFFTIME = 8

All other parameters can be set to 0.

2.3.11 GPMC_CONFIG5

In this section we will provide values for RDCYCLETIME, WRCYCLETIME and RDACCESSTIME.

RdAccessTime and CycleTime timing parameters configuration.

Figure 7-65. GPMC_CONFIG5_i

31 28	27 24	23 21	20 16
Reserved	PAGEBURSTACCESSTIME	Reserved	RDACCESSTIME
R-0	R/W-0	R-0	R/W-0

15 13	12 8	7 5	4 0
Reserved	WRCYCLETIME	Reserved	RDCYCLETIME
R-0	R/W-0	R-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

Table 7-69. GPMC_CONFIG5_i Field Descriptions

Bit	Field	Value	Description
31-28	Reserved	0	Reserved
27-24	PAGEBURSTACCESSTIME		Delay between successive words in a multiple access
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		Fh	15 GPMC_FCLK cycles
23-21	Reserved	0	Reserved
20-16	RDACCESSTIME		Delay between start cycle time and first data valid
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		1Fh	31 GPMC_FCLK cycles
15-13	Reserved	0	Reserved
12-8	WRCYCLETIME		Total write cycle time
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		1Fh	31 GPMC_FCLK cycles
7-5	Reserved	0	Reserved
4-0	RDCYCLETIME		Total read cycle time
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		1Fh	31 GPMC_FCLK cycles

Possible values for these parameters are:

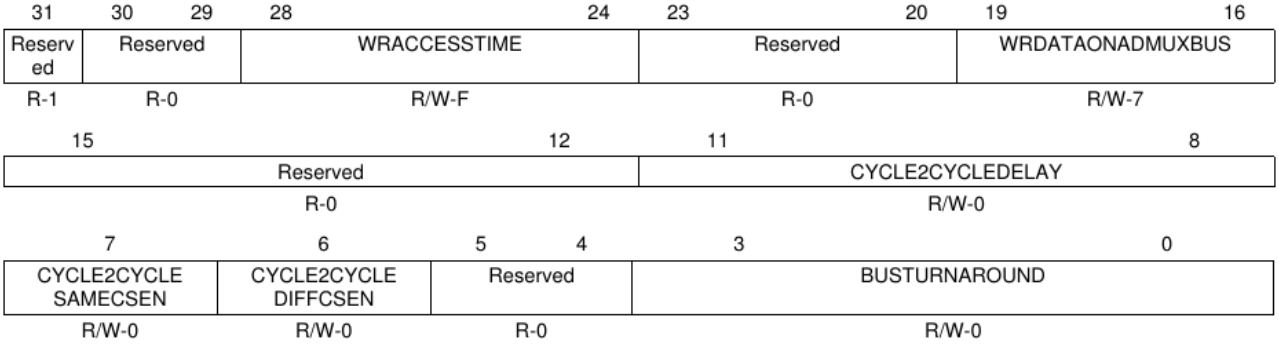
RDCYCLETIME = 10
WRCYCLETIME = 10
RDACCESSTIME = 8

All other parameters can be set to 0.

2.3.12 GPMC_CONFIG6

In this section we will configure WRDATAONADMUXBUS.

Figure 7-66. GPMC_CONFIG6_i



LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

Table 7-70. GPMC_CONFIG6_i Field Descriptions

Bit	Field	Value	Description
31-29	Reserved	1	Reserved
28-24	WRACCESSTIME		Delay from StartAccessTime to the GPMC.FCLK rising edge corresponding the the GPMC.CLK rising edge used by the attached memory for the first data capture
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		1Fh	31 GPMC_FCLK cycles
23-20	Reserved	0	Reserved
19-16	WRDATAONADMUXBUS	0-Fh	Specifies on which GPMC.FCLK rising edge the first data of the synchronous burst write is driven in the add/data multiplexed bus
15-12	Reserved	0	Reserved
11-8	CYCLE2CYCLEDELAY		Chip select high pulse delay between two successive accesses
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		Fh	15 GPMC_FCLK cycles
7	CYCLE2CYCLESAMECSEN		Add Cycle2CycleDelay between two successive accesses to the same chip-select (any access type)
		0	No delay between the two accesses
		1	Add CYCLE2CYCLEDELAY
6	CYCLE2CYCLEDIFFCSEN		Add Cycle2CycleDelay between two successive accesses to a different chip-select (any access type)
		0	No delay between the two accesses
		1	Add CYCLE2CYCLEDELAY
5-4	Reserved	0	Reserved
3-0	BUSTURNAROUND		Bus turn around latency between two successive accesses to the same chip-select (read to write) or to a different chip-select (read to read and read to write)
		0	0 GPMC_FCLK cycle
		1h	1 GPMC_FCLK cycle
		⋮	⋮
		Fh	15 GPMC_FCLK cycles

We may program WRDATAONADMUXBUS=4, and set all other parameters to 0.

2.3.13 GPMC_CONFIG7

Address decoder and Chip Select: we will configure the address set for which our CS* signal will be accessed.

First of the GPMC memory area of the microcontroller has been established at the beginning of the memory map:

Table 2-1. L3 Memory Map

Block Name	Start_address (hex)	End_address (hex)	Size	Description
GPMC (External Memory)	0x0000_0000[1]	0x1FFF_FFFF	512MB	8-/16-bit External Memory (Ex/R/W)[2]
Reserved	0x2000_0000	0x3FFF_FFFF	512MB	Reserved
Boot ROM	0x4000_0000	0x4001_FFFF	128KB	
	0x4002_0000	0x4002_BFFF	48KB	32-bit Ex/R[2] – Public
Reserved	0x4002_C000	0x400F_FFFF	848KB	Reserved
Reserved	0x4010_0000	0x401F_FFFF	1MB	Reserved
Reserved	0x4020_0000	0x402E_FFFF	960KB	Reserved
Reserved	0x402F_0000	0x402F_03FF	64KB	Reserved
SRAM internal	0x402F_0400	0x402F_FFFF		32-bit Ex/R/W[2]
L3 OCMC0	0x4030_0000	0x4030_FFFF	64KB	32-bit Ex/R/W[2] OCMC SRAM
Reserved	0x4031_0000	0x403F_FFFF	960KB	Reserved
Reserved	0x4040_0000	0x4041_FFFF	128KB	Reserved
Reserved	0x4042_0000	0x404F_FFFF	896KB	Reserved
Reserved	0x4050_0000	0x405F_FFFF	1MB	Reserved
Reserved	0x4060_0000	0x407F_FFFF	2MB	Reserved
Reserved	0x4080_0000	0x4083_FFFF	256KB	Reserved
Reserved	0x4084_0000	0x40DF_FFFF	5888KB	Reserved
Reserved	0x40E0_0000	0x40E0_7FFF	32KB	Reserved
Reserved	0x40E0_8000	0x40EF_FFFF	992KB	Reserved
Reserved	0x40F0_0000	0x40F0_7FFF	32KB	Reserved
Reserved	0x40F0_8000	0x40FF_FFFF	992KB	Reserved
Reserved	0x4100_0000	0x41FF_FFFF	16MB	Reserved
Reserved	0x4200_0000	0x43FF_FFFF	32MB	Reserved
L3F CFG Regs	0x4400_0000	0x443F_FFFF	4MB	L3Fast configuration registers

Therefore, the GPMC external memory then can be accessed at most in the address range 0x0000_0000 to 0x1FFF_FFFF.

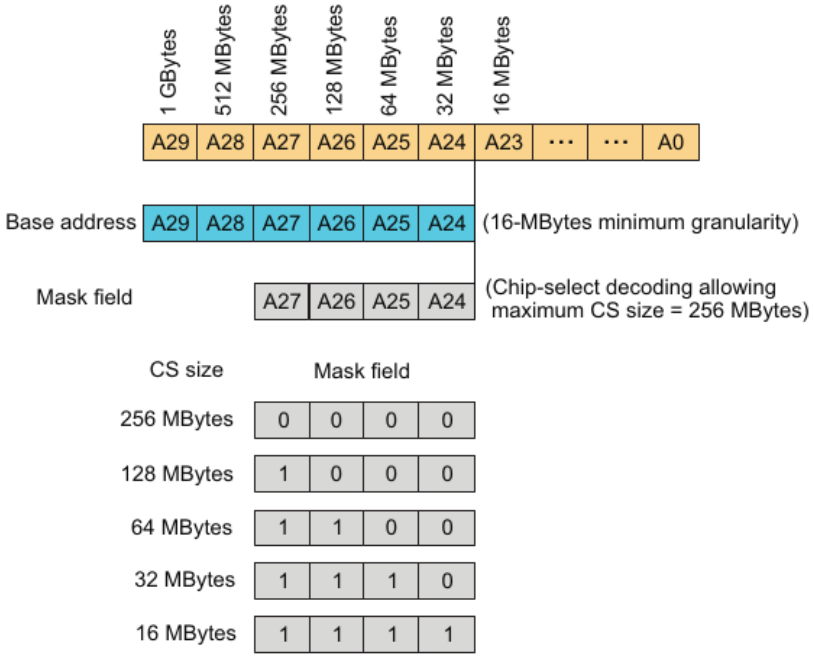
An important excerpt of the reference manual describes the generation of the address set for any CS:

The GPMC 512 Mbyte address space can be divided into a maximum of seven chip-select regions with programmable base address and programmable CS size. The CS size is programmable from 16 Mbytes to 256 Mbytes (must be a power-of-2) and is defined by the mask field. Attached memory smaller than the programmed CS region size is accessed through the entire CS region (aliasing).

Each chip-select has a 6-bit base address encoding and a 4-bit decoding mask, which must be programmed according to the following rules:

- The programmed chip-select region base address must be aligned on the chip-select region size address boundary and is limited to a power-of-2 address value. During access decoding, the register base address value is used for address comparison with the address-bit line mapping as described in Figure 7-6 (with A0 as the device system byte-address line). Base address is programmed through the GPMC_CONFIG7_i[5-0] BASEADDRESS bit field.
- The register mask is used to exclude some address lines from the decoding. A register mask bit field cleared to 0 suppresses the associated address line from the address comparison (incoming address bit line is don't care). The register mask value must be limited to the subsequent value, based on the desired chip-select region size. Any other value has an undefined result. When multiple chip-select regions with overlapping addresses are enabled concurrently, access to these chip-select regions is cancelled and a GPMC access error is posted. The mask field is programmed through the GPMC_CONFIG7_i[11-8] MASKADDRESS bit field.

Figure 7-6. Chip-Select Address Mapping and Decoding Mask



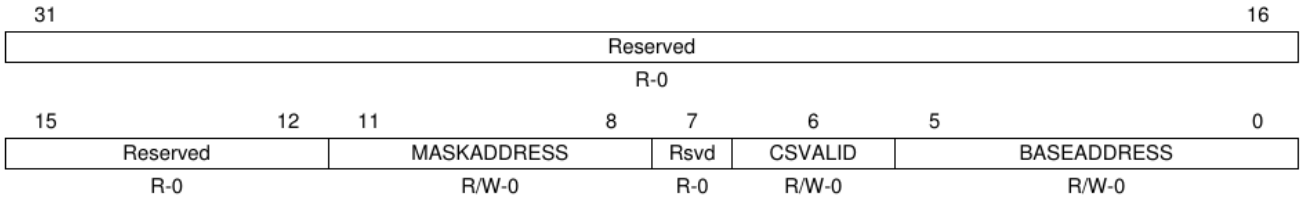
A mask value of 0010 or 1001 must be avoided because it will create holes in the chip-select address space.

Chip-select configuration (base and mask address or any protocol and timing settings) must be performed while the associated chip-select is disabled through the GPMC_CONFIG7_i[6] CSVALID bit. In addition, a chip-select configuration can only be disabled if there is no ongoing access to that chip-select. This requires activity monitoring of the prefetch or write-posting engine if the engine is active on the chip-select. Also, the write buffer state must be monitored to wait for any posted write completion to the chip-select.

The mask address is configured with the GPMC_CONFIG7 register, as well as the base address and finally the activation of the CS.

Chip-select address mapping configuration.

Figure 7-67. GPMC_CONFIG7_i



LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

Table 7-71. GPMC_CONFIG7_i Field Descriptions

Bit	Field	Value	Description
31-12	Reserved	0	Reserved
11-8	MASKADDRESS		Chip-select mask address. Values not listed must be avoided as they create holes in the chip-select address space.
		0	Chip-select size of 256 Mbytes
		8h	Chip-select size of 128 Mbytes
		Ch	Chip-select size of 64 Mbytes
		Eh	Chip-select size of 32 Mbytes
		Fh	Chip-select size of 16 Mbytes
7	Reserved	0	Reserved
6	CSVALID		Chip-select enable (reset value is 1 for $\overline{CS}[0]$ and 0 for $\overline{CS}[1\text{-}5]$).
		0	$\overline{CS}$ disabled
		1	$\overline{CS}$ enabled
5-0	BASEADDRESS	0-3Fh	Chip-select base address.
			CSi base address where i = 0 to 3 (16 Mbytes minimum granularity). Bits [5-0] corresponds to A29, A28, A27, A26, A25, and A24.

Configure this register so that the access to the GPMC device, using CS2*, is made from the address 0x0900_0000 onwards.

OPTIONAL:

Finally we may produce the L3 clock / 5 = 40 MHz in CLKOUT2:

```
hwreg(soc_cm_device_base+CM_CLKOUT_CTRL) = 0xA1;
```

2.4 Pin Configuration: MODE 0, Input Enable and PULL UP

In page 157 of the reference manual (version H) we find that the following address range is reserved for the Control Module that controls the pin configuration.

	0x44E0_A000	0x44E0_AFFF	4KB	Reserved
I2C0	0x44E0_B000	0x44E0_BFFF	4KB	I2C Registers
	0x44E0_C000	0x44E0_CFFF	4KB	Reserved
ADC_TSC	0x44E0_D000	0x44E0_EFFF	8KB	ADC_TSC Registers
	0x44E0_F000	0x44E0_FFFF	4KB	Reserved
Control Module	0x44E1_0000	0x44E1_1FFF	128KB	Control Module Registers
DDR2/3/mDDR PHY	0x44E1_2000	0x44E1_23FF		DDR2/3/mDDR PHY Registers
Reserved	0x44E1_2400	0x44E3_0FFF	4KB	Reserved
DMTIMER1_1MS (Accurate 1ms timer)	0x44E3_1000	0x44E3_1FFF	4KB	DMTimer1 1ms Registers

Now, the addresses of the control registers for each pin can be accessed as:

```
address pin control register = 0x44E1_0000 + OFFSET registers
```

The offsets of the registers you have to configure can be found in page 758 of the reference manual (rev. H) or page 1124 (rev. I):

```
#define GPMC_AD0_OFFSET                 0x800
#define GPMC_AD1_OFFSET                 0x804
#define GPMC_AD2_OFFSET                 0x808
#define GPMC_AD3_OFFSET                 0x80C
#define GPMC_AD4_OFFSET                 0x810
#define GPMC_AD5_OFFSET                 0x814
#define GPMC_AD6_OFFSET                 0x818
#define GPMC_AD7_OFFSET                 0x81C
#define GPMC_AD8_OFFSET                 0x820
#define GPMC_AD9_OFFSET                 0x824
#define GPMC_AD10_OFFSET                0x828
#define GPMC_AD11_OFFSET                0x82C
#define GPMC_AD12_OFFSET                0x830
#define GPMC_AD13_OFFSET                0x834
#define GPMC_AD14_OFFSET                0x838
#define GPMC_AD15_OFFSET                0x83C
#define GPMC_CSN2_OFFSET                0x884
#define GPMC_ADVn_ALE_OFFSET            0x890
#define GPMC_BE0n_CLE_OFFSET            0x89C
#define GPMC_BE1n_OFFSET                0x878
#define GPMC_CLK_OFFSET                 0x88C
#define GPMC_OE_REn_OFFSET              0x894
#define GPMC_WEn_OFFSET                 0x898
```

You can use the above labels to access all the configuration registes of the pins we will be using. The description of the bit fields of the above registers is the following:

9.3.51 conf_<module>_<pin> Register (offset = 800h–A34h)

See the device datasheet for information on default pin mux configurations. Note that the device ROM may change the default pin mux for certain pins based on the SYSBOOT mode settings.

See Table 9-10, *Control Module Registers Table*, for the full list of offsets for each module/pin configuration.

conf_<module>_<pin> is shown in Figure 9-54 and described in Table 9-61.

Figure 9-54. conf_<module>_<pin> Register

31	30	29	28	27	26	25	24
Reserved							
R-0h							

23	22	21	20	19	18	17	16
Reserved				Reserved			
R-0h				R-0h			

15	14	13	12	11	10	9	8
Reserved							
R-0h							

7	6	5	4	3	2	1	0
Reserved	conf_<module>_<pin>_slewctrl	conf_<module>_<pin>_rxactive	conf_<module>_<pin>_putypesel	conf_<module>_<pin>_puden	conf_<module>_<pin>_mmode		
R-0h	R/W-0h	R/W-1h	R/W-0h	R/W-0h	R/W-0h		

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -*n* = value after reset

Table 9-61. conf_<module>_<pin> Register Field Descriptions

Bit	Field	Type	Reset	Description
31-20	Reserved	R	0h	
19-7	Reserved	R	0h	
6	conf_<module>_<pin>_slewctrl	R/W	X	Select between faster or slower slew rate 0: Fast 1: Slow Reset value is pad-dependent.
5	conf_<module>_<pin>_rxactive	R/W	1h	Input enable value for the PAD 0: Receiver disabled 1: Receiver enabled
4	conf_<module>_<pin>_putypesel	R/W	X	Pad pullup/pulldown type selection 0: Pulldown selected 1: Pullup selected Reset value is pad-dependent.
3	conf_<module>_<pin>_puden	R/W	X	Pad pullup/pulldown enable 0: Pullup/pulldown enabled 1: Pullup/pulldown disabled Reset value is pad-dependent.
2-0	conf_<module>_<pin>_mmode	R/W	X	Pad functional signal mux select. Reset value is pad-dependent.

We will configure all the above 23 pins to the FAST, RECEIVER ENABLED, PULL-UP Enabled and MODE 0.

3 Guided configuration

The previous section contains large quantities of information that in general are difficult to assimilate. In order to ease out the task we propose to you, you may use the following programming instructions:

3.1 Function void pointers_init(void)

This function must give values to the following pointers that must be defined as global variables:

```
static void __iomem *soc_device_id;
static void __iomem *soc_gpmc_base;
static void __iomem *soc_prcm_base;
static void __iomem *soc_cm_device_base;
static void __iomem *soc_extmem_base;
static void __iomem *soc_pin_mux_base;
```

The idea is to have a different pointer, living in the kernel virtual memory space, with a value correctly assigned by the MMU to the corresponding physical address. As an example we have in our initial driver:

```
soc_device_id = ioremap(SOC_DEVICE_ID_REGS, 4);
```

As we have previously seen the SOC_DEVICE_ID_REGS is a label defined in the soc_AM335.h header file, with the value (0x44E10600). You should check in page 163 of the datasheet of the microprocessor that this is indeed the address of the identification register of the microprocessor.

The values that must be assigned to each pointer must be obtained by calling to the function ioremap with the following parameters for each pointer.

Kernel pointer	Address label (physical address value)	SIZE of the part of memory assigned to each pointer
soc_extmem_base	EXTMEM_BASE	SZ_128K
soc_gpmc_base	SOC_GPMC_0_REGS	SZ_16M
soc_prcm_base	SOC_PRCM_REGS	0x12FF
soc_cm_device_base	SOC_CM_DEVICE_REGS	SZ_256
soc_pin_mux_base	AM33XX_CTRL_BASE	PAD_CTRL_SIZE
soc_device_id	SOC_DEVICE_ID_REGS	4

The size labels are standard and be used within the file module.c. The only one that has been defined for this case is PAD_CTRL_SIZE that can be found in driver.h, with value 0xA38.

The first pointer, ***static void_iomem *soc_extmem_base***, is very important because it will be constantly used to access the block of memory assigned to the GPMC bus. Since we will conneed AD0-AD15 to the FPGA, this memory block is 128 Kbytes long.

3.2 Function uint gpmc_configuration(void)

This function must properly configure all the registers that are needed to specify how the GPMC must work. Since this means to program many different registers we propose to use the following steps.

Step 1: enable clock to GPMC module

Content addressed by	OP	VALUE
oc_prcm_base + CM_PER_GPMC_CLKCTRL	| =	CM_PER_GPMC_CLKCTRL_MODULEMODE_ENABLE

Step 2: Wait till GPMC module is enabled

```
while((hwreg(soc_prcm_base + CM_PER_GPMC_CLKCTRL) &
            CM_PER_GPMC_CLKCTRL_IDLEST) !=
            (CM_PER_GPMC_CLKCTRL_IDLEST_FUNC <<
            CM_PER_GPMC_CLKCTRL_IDLEST_SHIFT));
```

Step 3: RESET the GPMC module

Content addressed by	OP	VALUE
soc_gpmc_base + GPMC_SYSCONFIG	| =	GPMC_SYSCONFIG_SOFTRESET

Step 4: Wait till RESET DONE

```
while((hwreg(soc_gpmc_base + GPMC_SYSSTATUS) & GPMC_SYSSTATUS_RESETDONE)
            ==  GPMC_SYSSTATUS_RESETDONE_RSTONGOING);
```

Step 5: Configure to no idle (define an auxiliary variable (uint) temp)

```
temp = hwreg(soc_gpmc_base + GPMC_SYSCONFIG);
temp &= ~GPMC_SYSCONFIG_IDLEMODE;
temp |= GPMC_SYSCONFIG_IDLEMODE_NOIDLE <<
                                    GPMC_SYSCONFIG_IDLEMODE_SHIFT;
hwreg(soc_gpmc_base + GPMC_SYSCONFIG) = temp;
hwreg(soc_gpmc_base + GPMC_IRQENABLE) = 0x0;
hwreg(soc_gpmc_base + GPMC_TIMEOUT_CONTROL) = 0x0;
```

Step 6: configure for NOR and granularity x2

CONFIG 1: 16 bits multiplexed

Content addressed by	OP	VALUE
soc_gpmc_base + GPMC_CONFIG1(csNum)	=	(0x0 | (GPMC_CONFIG1_0_DEVICESIZE_SIXTEENBITS << GPMC_CONFIG1_0_DEVICESIZE_SHIFT) | (GPMC_CONFIG1_0_ATTACHEDDEVICEPAGELENGTH_FOUR << GPMC_CONFIG1_0_ATTACHEDDEVICEPAGELENGTH_SHIFT) | (0x2 << 8))

Step 7: chip select assert/deassert times

CONFIG 2

Content addressed by	OP	VALUE
soc_gpmc_base + GPMC_CONFIG2(csNum)	=	(0x0 | (0x01) | (0xa << GPMC_CONFIG2_0_CSRDOFFTIME_SHIFT) | (0xa << GPMC_CONFIG2_0_CSWROFFTIME_SHIFT))

Step 8: chip select assert/deassert times

CONFIG 3

Content addressed by	OP	VALUE
soc_gpmc_base + GPMC_CONFIG3(csNum)	=	(0x0 | (1 << GPMC_CONFIG3_0_ADVONTIME_SHIFT) | (3 << GPMC_CONFIG3_0_ADVRDOFFTIME_SHIFT) | (3 << GPMC_CONFIG3_0_ADVWROFFTIME_SHIFT))

Step 9: output enable / read write enable assert and de-assert

CONFIG 4

Content addressed by	OP	VALUE
soc_gpmc_base + GPMC_CONFIG4(csNum)	=	(0x0 | (0x5 << GPMC_CONFIG4_0_OEONTIME_SHIFT) | (0x9 << GPMC_CONFIG4_0_OEOFFTIME_SHIFT) | (0x5 << GPMC_CONFIG4_0_WEONTIME_SHIFT) | (0x8 << GPMC_CONFIG4_0_WEOFFTIME_SHIFT))

Step 10: read and write cycle time

CONFIG 5

Content addressed by	OP	VALUE
soc_gpmc_base + GPMC_CONFIG5(csNum)	=	(0x0 | (0xa << GPMC_CONFIG5_0_RDCYCLETIME_SHIFT) | (0xa << GPMC_CONFIG5_0_WRCYCLETIME_SHIFT) | (0x8 << GPMC_CONFIG5_0_RDACCESSTIME_SHIFT))

Step 11: bus turnaround delay, etc .

CONFIG 6:

Content addressed by	OP	VALUE
soc_gpmc_base + GPMC_CONFIG6(csNum)	=	(0x0 | (0 << GPMC_CONFIG6_0_CYCLE2CYCLESAMECSEN_SHIFT) | (0 << GPMC_CONFIG6_0_CYCLE2CYCLEDELAY_SHIFT) | (4 << GPMC_CONFIG6_0_WRDATAONADMUXBUS_SHIFT) | (0 << GPMC_CONFIG6_0_WRACCESSTIME_SHIFT))

Step 12: base address of chip select and address space (16 MB or more)

CONFIG 7:

Content addressed by	OP	VALUE
soc_gpmc_base + GPMC_CONFIG7(csNum)	=	(0x09 << GPMC_CONFIG7_0_BASEADDRESS_SHIFT) | (0x1 << GPMC_CONFIG7_0_CSVALID_SHIFT) | (0x0f << GPMC_CONFIG7_0_MASKADDRESS_SHIFT)

Step 13: Send L3 clock/5 = 40 Mhz to CLKOUT2

Content addressed by	OP	VALUE
soc_cm_device_base+CM_CLKOUT_CTRL	=	0xA1

3.3 Function int pin_mux_configuration(void)

This function must configure adequately the register that specifies which mode will use the pins involved in the GPMC bus that we have connected to the FPGA. In this regard the set of operations is summarised in the following table:

Content addressed by pointer: soc_pin_mux_base + OFFSET	OP	VALUE
GPMC_AD0_OFFSET GPMC_AD1_OFFSET GPMC_AD2_OFFSET GPMC_AD3_OFFSET GPMC_AD4_OFFSET GPMC_AD5_OFFSET GPMC_AD6_OFFSET GPMC_AD7_OFFSET GPMC_AD8_OFFSET GPMC_AD9_OFFSET GPMC_AD10_OFFSET GPMC_AD11_OFFSET GPMC_AD12_OFFSET GPMC_AD13_OFFSET GPMC_AD14_OFFSET GPMC_AD15_OFFSET GPMC_CSN2_OFFSET GPMC_ADVn_ALE_OFFSET GPMC_BE0n_CLE_OFFSET GPMC_BE1n_OFFSET GPMC_CLK_OFFSET GPMC_OE_REn_OFFSET GPMC_WEn_OFFSET	=	OMAP_MUX_MODE0 | AM33XX_INPUT_EN | AM33XX_PULL_UP

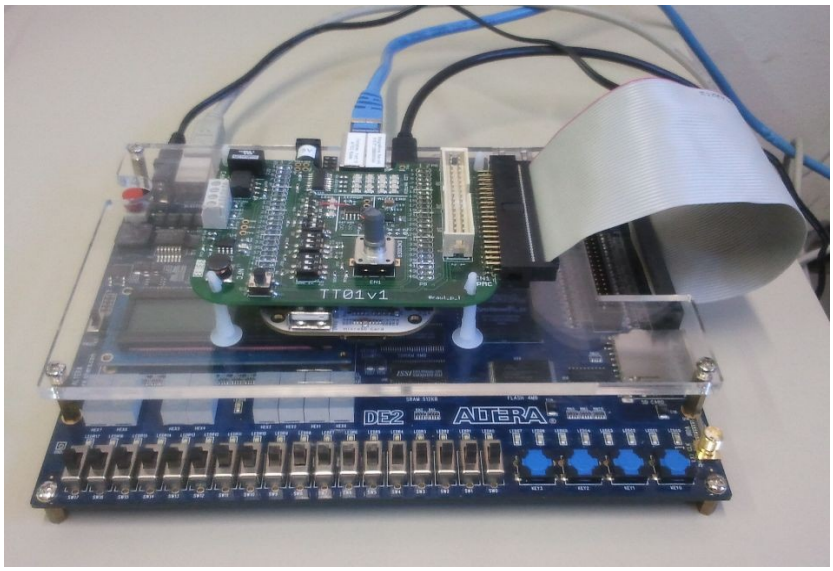
Once again, the labels OMAP_MUX_MODE0, etc. are defined in the file *driver.h*.

4 Laboratory Work

4.1 Mandatory work

1. In order to work with the same set of files and procedures as in the example given in section 2, create a copy of the directory `lm11` and name it `lm11b`. This will be the new working directory.

2. Following the information provided in section 3, configure and compile `module.c` in order to implement the GMPC functionality required. Check the console messages using the `screen` utility and also with `dmesg` (this last instruction is executed on the Beaglebone).



3. Ask your Lab Instructor for the DE2 board and connect it to the Beaglebone cape and to a power source. As commented in previous sections, the DE2 FPGA will be programmed to work as a bank of 8 registers of 16 bits each one, with consecutive addresses.

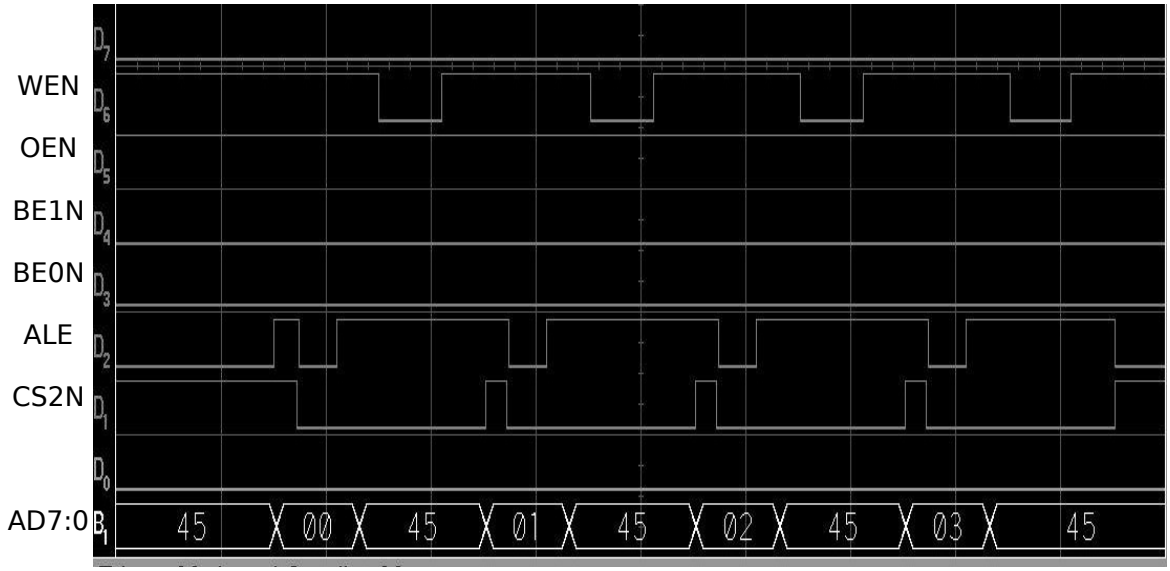
This means that there are 16 bytes of continuous memory in the FPGA. They can be accessed from address:

```
*soc_extmem_base + 0x00000    to    *soc_extmem_base + 0x000010
```

4. Configure and compile `app.c` in order to obtain an application that uses the GPC interface to write and read data to/from the latches of the DE2 FPGA. The application must be able to write and then read all 16 bytes in the FPGA.

5. Execute the application. The data send or received must be displayed both in the Beaglebone terminal and in the DE2 displays.

6. You must also be able to see the waveforms of the signals using the logic analyzer, as in the following example:



In the above example we see a sequence of write cycles to consecutives address (0x00, 0x01, 0x02 and 0x03) where the data being written in the low part of the data bus is always 0x45.

4.2 Optional Work

1. It is possible to use the ioctl system call to send commands to the driver. This means that a call in the application side as:

```
ioctl(f,command,value);
```

may be used by a routine like the following, to give a certain value to a global variable:

```
static long sd_ioctl( struct file* pFile, unsigned int cmd, unsigned long value)
{

        if (cmd==1) {
                variable1 = value;
        } else if (cmd==2) {
                variable2 = value;
        }
        // and so on

        return 0;
}
```

You can check this.

2. There are kernel semaphores. All the access to the GPMC bus can be controlled with one of them to avoid concurrency problems. You can use them as in the following example:

```
#include <linux/semaphore.h>

struct semaphore mysem;
....

if (down_interruptible(&mysem))
                return -ERESTARTSYS;

// CRITICAL SECTION

up (&mysem);


sema_init(&mysem, 1);  // Initialization of the semaphore
```

The *down_interruptible* instruction attempts to the decrease the semaphore value. In case the value of the semaphore is zero then it puts to the process to sleep until it can be decreased without taking negative values. The critical section in our case may be the access to the GPMC bus in the read and write functions.